15-453 FLAC
Lecture 19

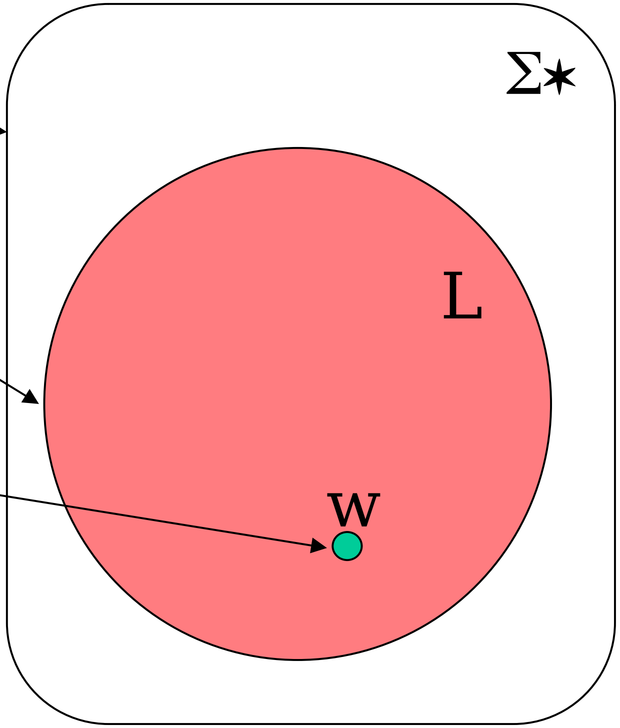
Turing Machines and Real Life
*
Reductions

Mihai Budiu

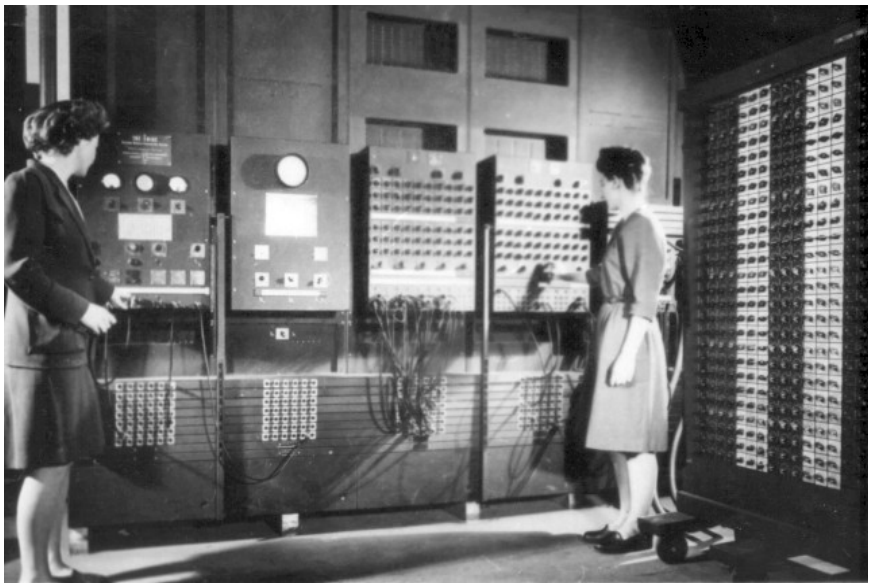
March 3, 2000

A Turing Machine solves one problem

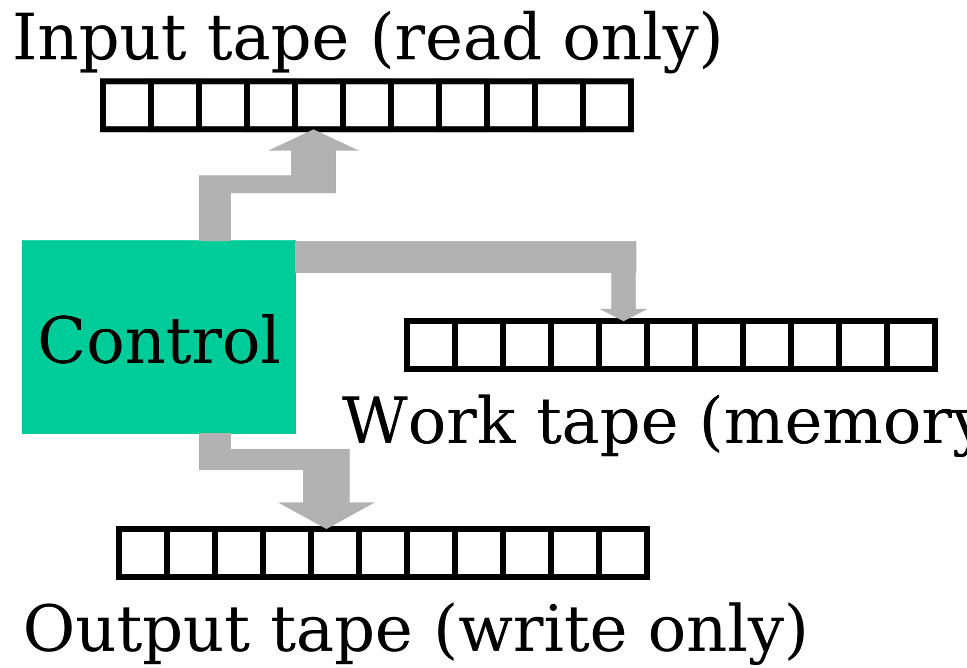
- We must distinguish between
  - The set of all strings
  - A problem (e.g. the language to be accepted)
  - A problem instance (e.g. a word from the language)
- A TM solves each instance presented as input



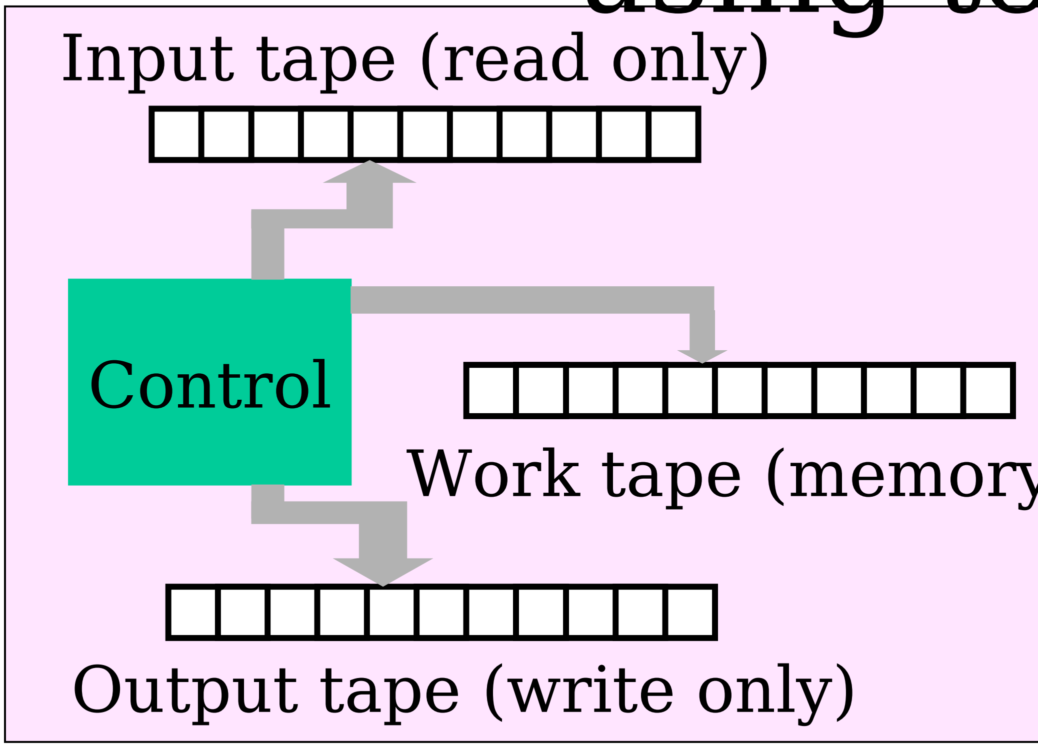
First computers: custom computing machines



1950 -- Eniac: the control is hardwired manually for each problem.



TMs can be described using text



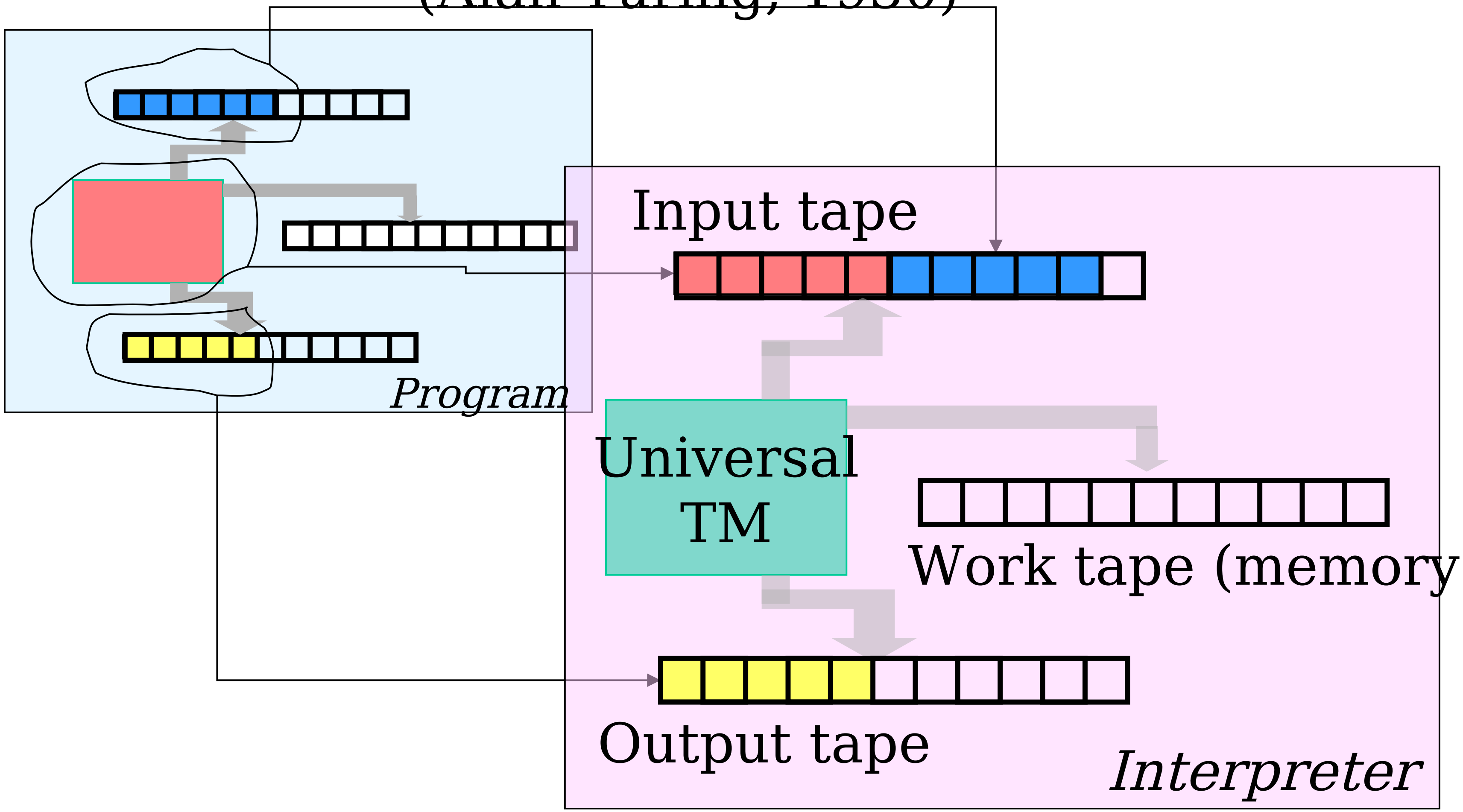
Program

- n states
- s letters for alphabet
- transition function:
  - d(q0,a) = (q1, b, L)
  - q1, a,

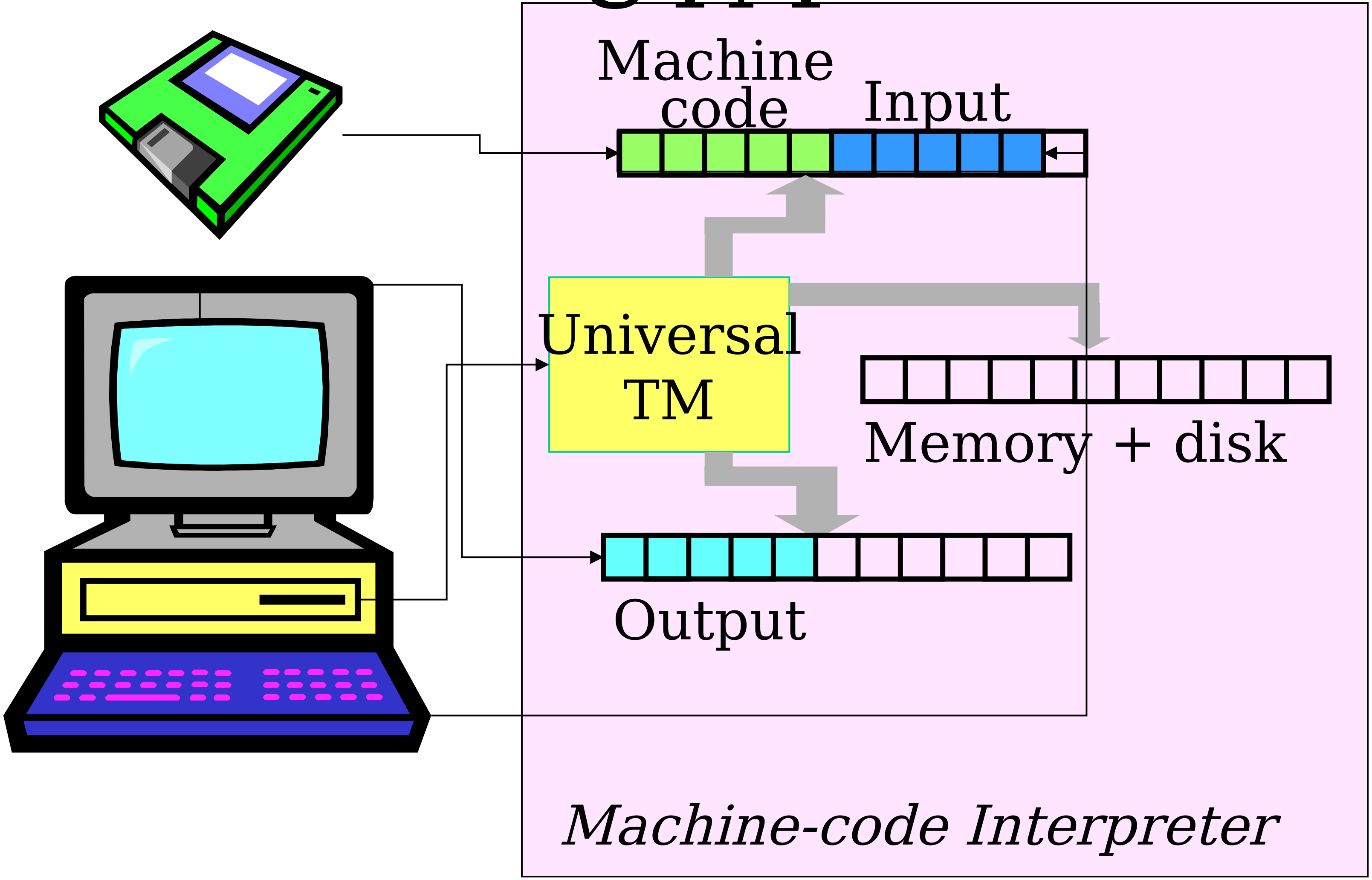
Consequence:
There is a *countable* number of Turing Machines

There is a TM which can simulate any other TM

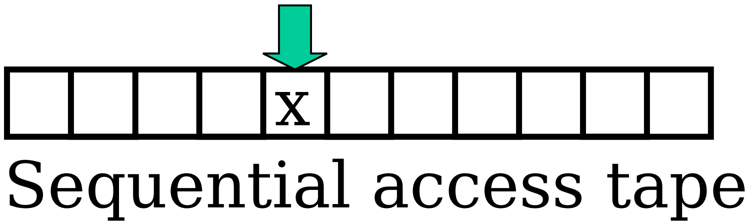
(Alan Turing, 1930)

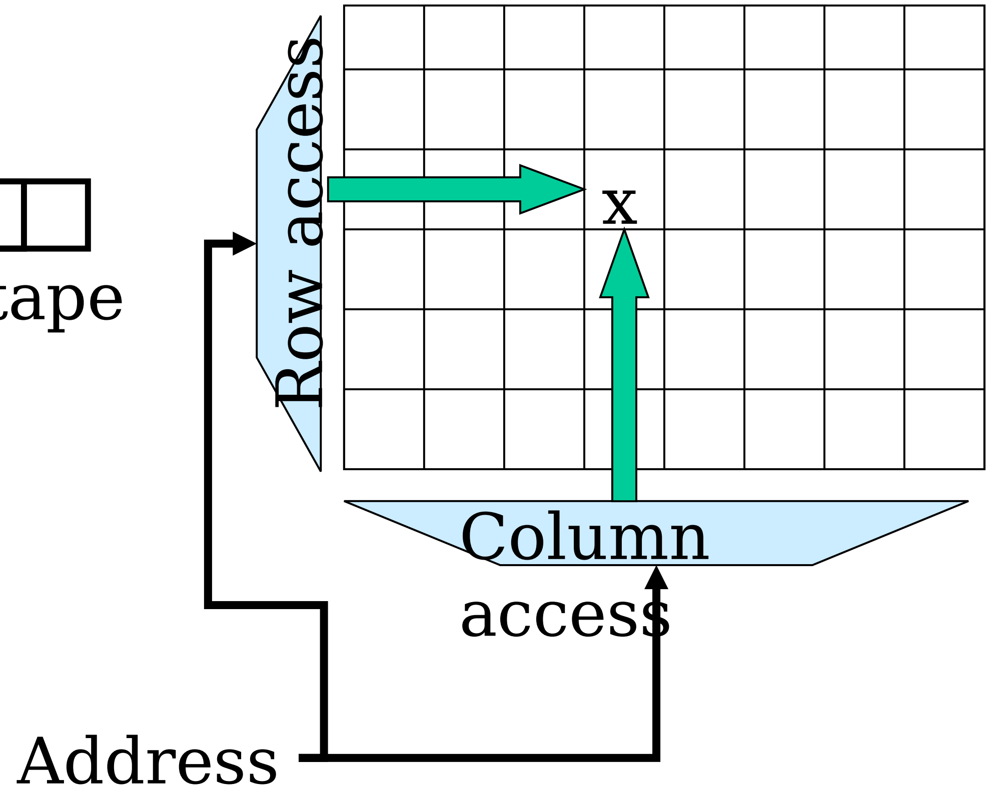


The Digital Computer: a UTM

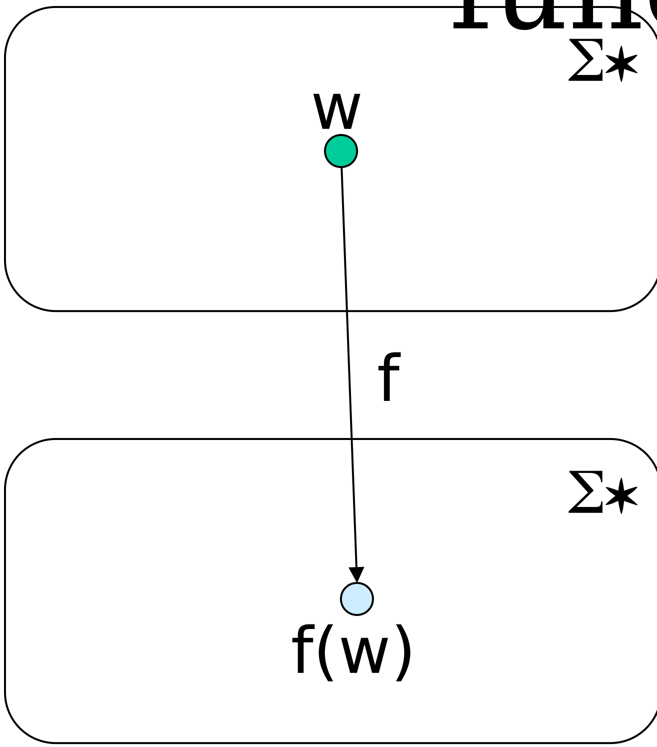


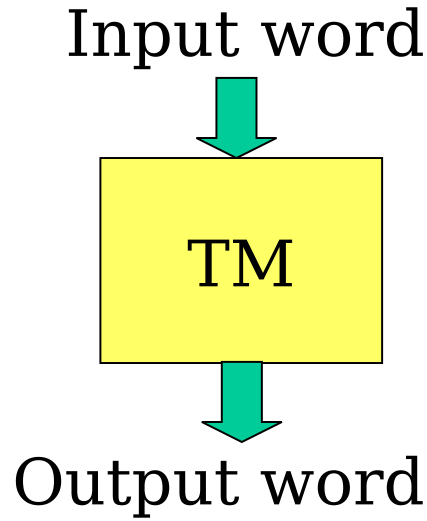
Random Access Memories





Turing machines as functions



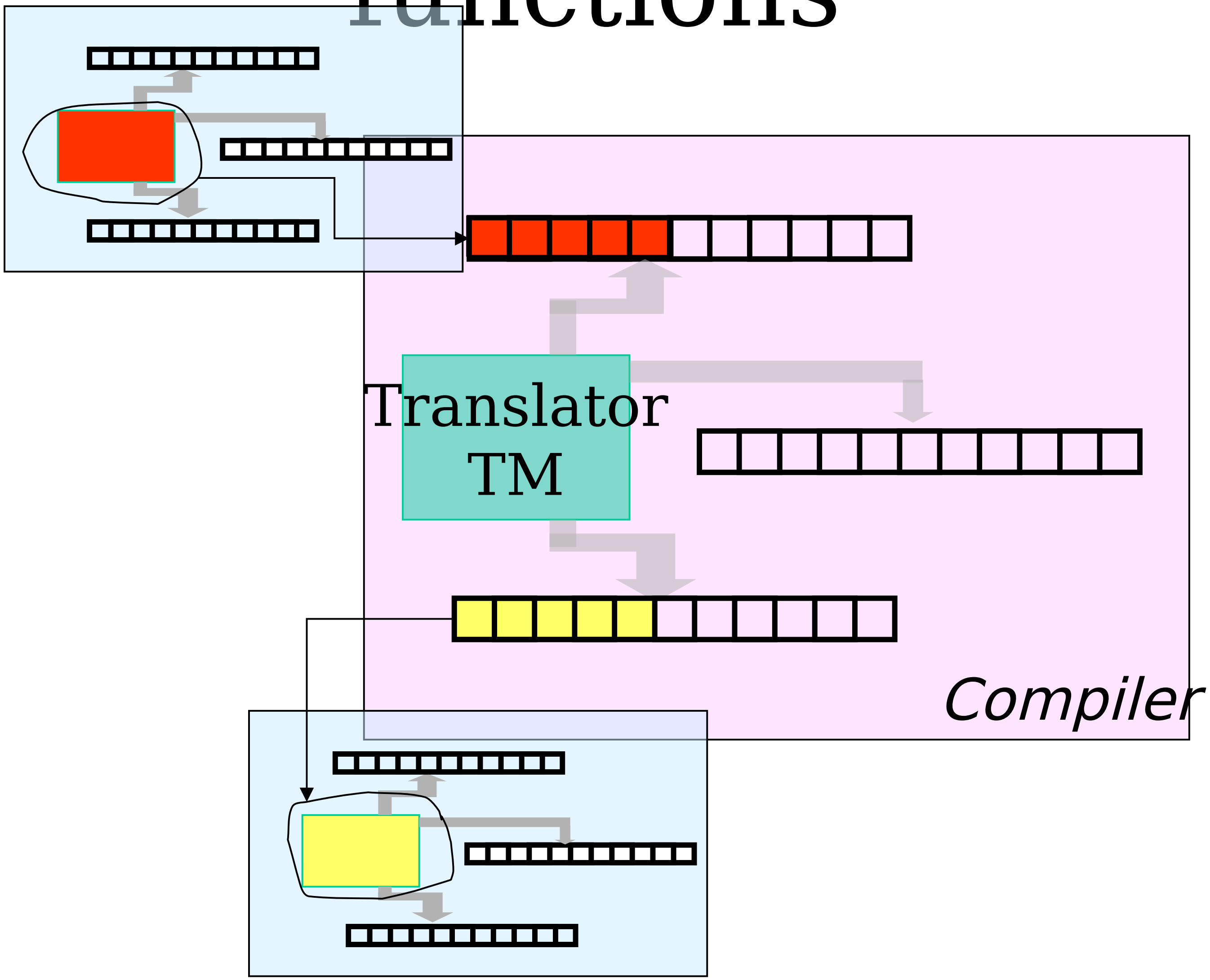


A function f is a *computable function* if a TM with word w as input *halts* with f(w) as output.

Computable functions

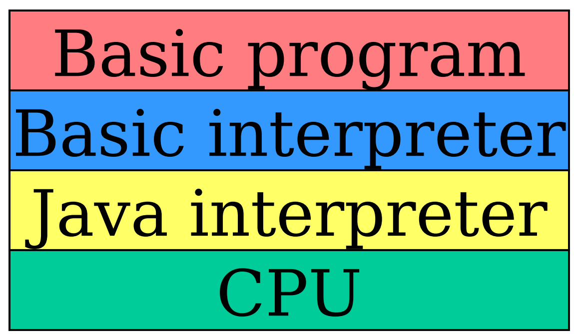
- Many functions encountered in everyday life are computable:
  - Addition : binary strings * binary strings -> binary strings
  - Sort : sequence of strings -> sequence of strings
  - Roots : polynomial -> sequence of integer roots [pg 144]
- A TM which decides a language computes a function from the set of all words to {y,n}
- There are many uncomputable functions:
  - |funs : N -> N| = uncountable
  - |TMs| = countable

Compilers: computable functions

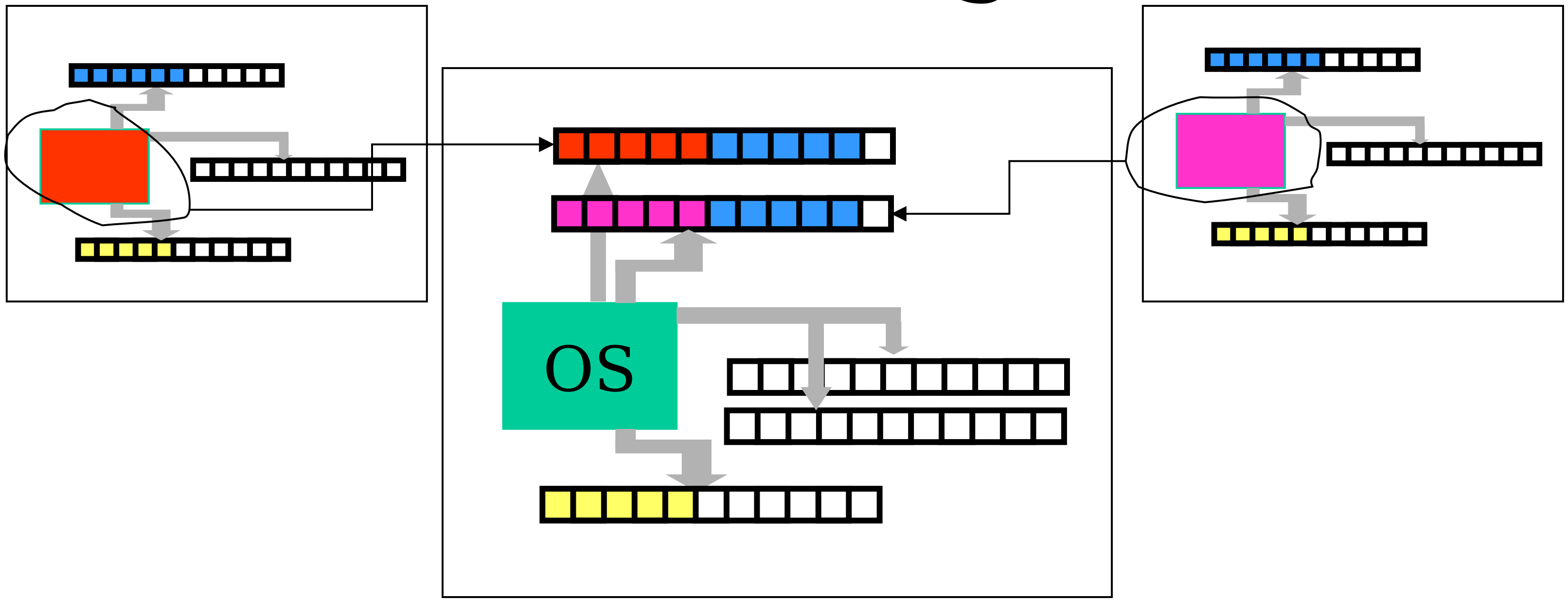


Multiple levels of virtualization

Ex: running a BASIC interpreter written in Java



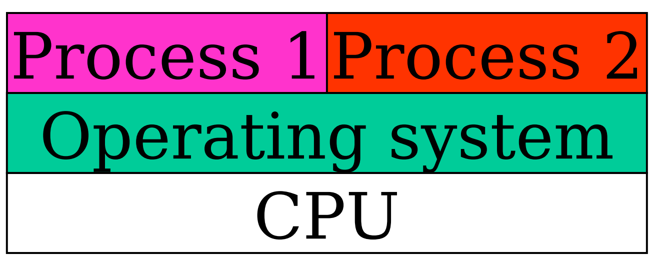
The operating system: another (multi-tape) universal turing machine



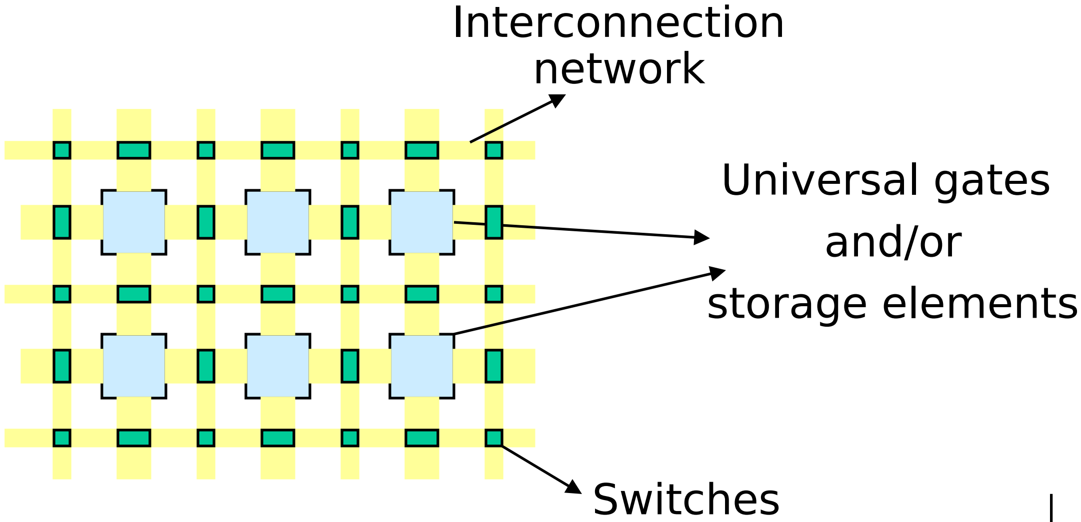
Process 1	Process 2
Operating system	
CPU	

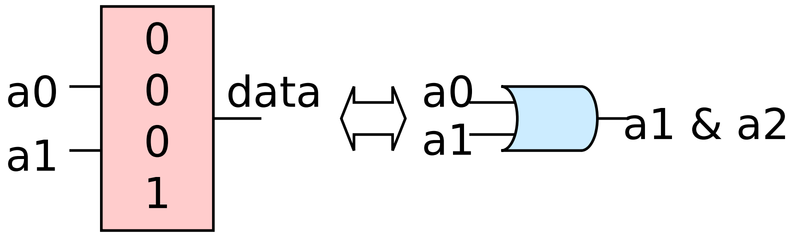
The operating system = dovetailing

- We used this technology in the proof of the last theorem during the last lecture:
- **Theorem:** L is decidable if and only if
  L and $\sim L$ are recognizable.
- **Proof:** execute in parallel recognizers for L and $\sim L$
  and stop when the first stops.

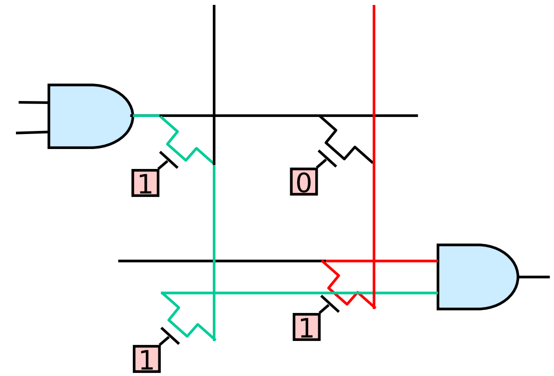


Reconfigurable Hardware





Universal gate = RAM



A switch is controlled by a 1-bit RAM cell

Reconfigurable hardware

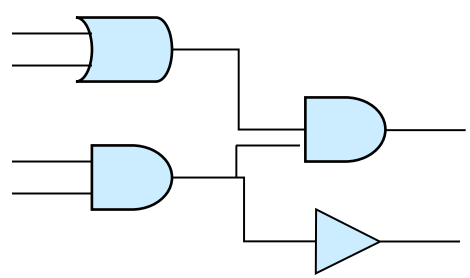
Program

Compiler

Custom hardware (configuration)

If (a) c = 2;

Compiler



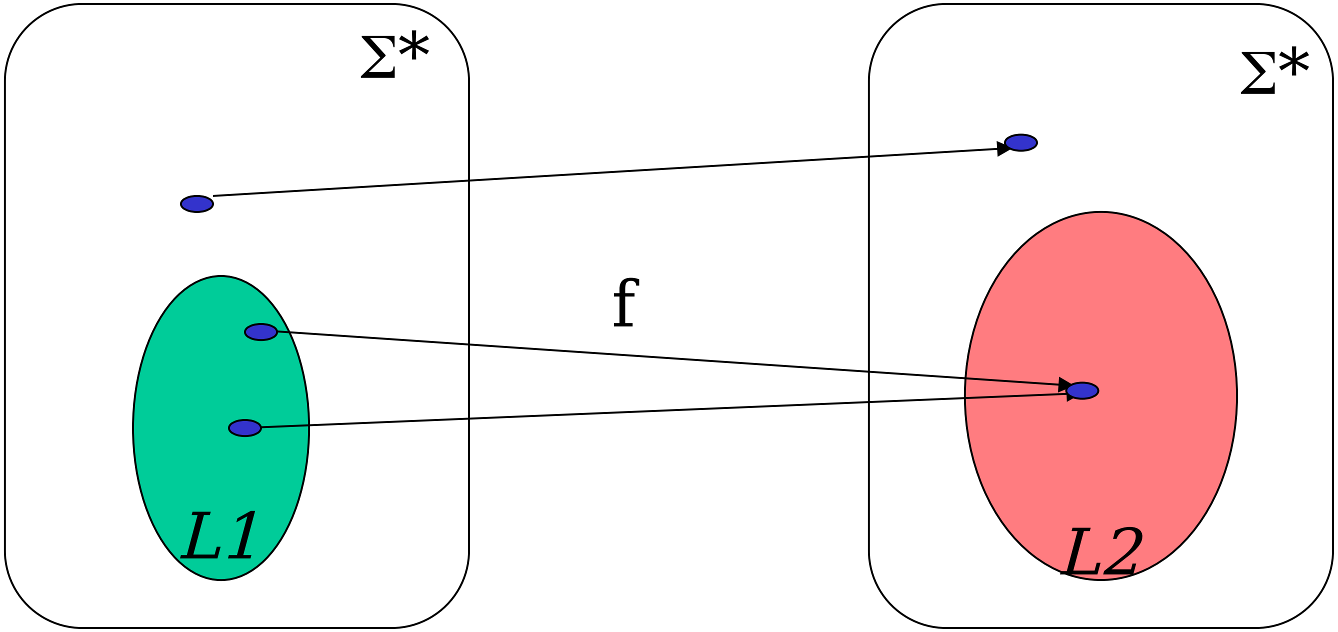
Generate a TM for a specific problem! Back to ENIAC!

Reductions between languages

f : Σ* -> Σ*, **computable**

f reduces *L1* to *L2 iff*

w in L1 <=> f(w) in L2



f is called **reduction.** We write $L1 \leq_m L2$.

Reductions are useful

Notice that neither *L1* or *L2* must be decidable.

Theorem: if $L1 \leq_m L2$ and *L2* is decidable then *L1* is decidable.

Proof: If M decides *L2* and N computes a reduction f from *L1* to *L2*, build a TM P which on input w:

- Computes f(w) using N
- Runs M on f(w)
- Outputs the result of M.

Using reductions to prove undecidability

- Use the converse of the previous theorem:
  if $A \leq_m B$ and A is undecidable, then B is undecidable.
- We have seen two techniques to prove undecidability
  - The “diagonalization” proof
  - Reductions using this theorem.

Undecidability

- We know that most problems are undecidable
- Turing exhibited one natural undecidable problem: the Halting Problem (does a TM halt when given an input w?)
- More than that: many important problems are undecidable
- When you face a problem, you should be aware that it may be unsolvable:
  - You can search a solution
  - You can try to prove it has no solution (usu. by reduction)

Some undecidable problems

- Does the TM M accept the word w?
- Is a mathematical statement true?
- Does a Diofantine equation have any roots?
- Does a TM accept a regular language?
- Does a CFG generate all words in Σ^*?
- Will a parallel system of processes ever deadlock?
- Is there a smaller TM implementing the same function?
- Are two programs computing the same thing?

Compilers and undecidability

- The HP is about a language describing TMs
- Many other problems concerning TMs are undecidable
- Compilers cannot prove some properties of programs:
  - Does a C program access an array out of bounds?
  - Will ever a print instruction be executed?
  - Can these two pointers point to the same location?
  - Will a LISP program crash with a type error?
  - Can this memory location be garbage collected at some point?
- Compilers behave conservatively

The full-employment theorem for compiler-writers

Theorem: There is no best optimizing compiler for a general-purpose language.

Proof: by reduction to the HP.

Theorem: For any compiler, there's a better one.

Proof: we can always hardcode a program which doesn't terminate.

Research in programming languages

- Contrast the decision problems for regular and context-free languages with Turing machines.
- Research in programming languages tries to get the best of both worlds:
  - Languages powerful enough to express useful computations
  - But restricted enough to guarantee program properties

Reduction examples

Definition: FIN = { M | L(M) is finite }

Theorem:

FIN is not Turing-recognizable.

~FIN is not Turing-recognizable.

Proof: We will show

$\sim HP \leqslant_m FIN$

$\sim HP \leqslant_m \sim FIN$

Notice that

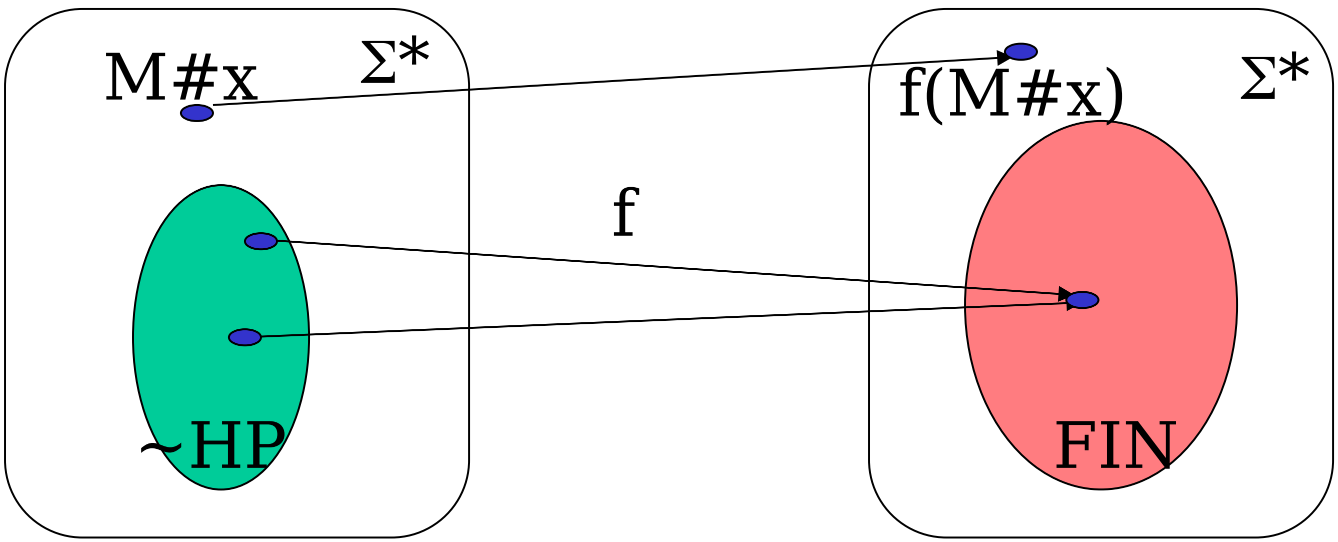
HP is Turing-recognizable

HP is not Turing-decidable

So

~HP is not Turing-recognizable.

A reduction from ~HP to FIN



f(M#x) is the description of a TM M’ which on input y:

- Erases the input y
- Writes x on the input tape
- Runs M on x
- Accepts if M halts on x

Reducing ~HP to FIN (end)

f(M#x) is the description of a TM M' which on input y:

- Erases the input y
- Writes x on the input tape
- Runs M on x
- accepts if M halts on x

Notice that

M halts on x ⇨ L(M') = Σ^*

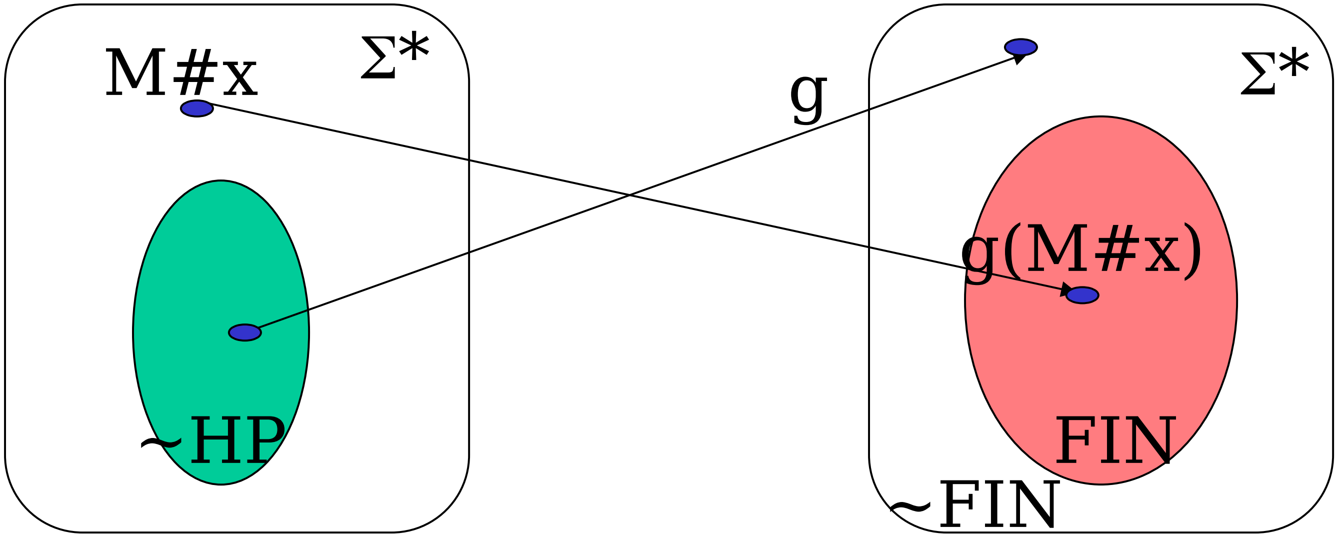
M does not halt on x ⇨ L(M') = Φ (finite)

Notice that f is computable:

f does not run M,

f just writes down a machine M' which calls M on

Part 2: Reducing ~HP to ~FIN



g(M#x) is the description of a TM M'' which on input y:

- Saves y
- Writes x on the input tape
- Runs M on x for |y| steps
- Accepts if M does not halt before |y| steps

Reducing ~HP to ~FIN (end)

g(M#x) is the description of a TM M’’ which on input y:

- Saves y
- Writes x on the input tape
- Runs M on x for |y| steps
- Accepts if M does not halt before |y| steps

M halts on x $\Rightarrow$ L(M’’) = { y | |y| < run-time of M on input x } finite

M does not halt on x $\Rightarrow$ L(M’’) = Σ^*

Again: g is a computable function
it just manipulates machine descriptions.

To remember

- The theoretical notion of TM has profoundly influenced computer engineering
- There are many important undecidable problems
- Reductions are a tool to prove problems being undecidable

  (Note: we will see reductions again in complexity theory; we will use them to prove a problem is the hardest in a class of problems)